2021

DOSSIER DE CANDIDATURE

APPLICATION

Cochez le concours sur lequel vous candidatez
Check the competition exam for which you are applying

☒ **ISFP - (Inria Starting Faculty Position / Inria Starting Faculty Position)**

☒ **CRCN - (Chargés de recherche de classe normale / Young graduate scientist position)**

☐ **DR2 - (Directeurs de recherche de deuxième classe / Senior researcher position)**

Nom[1] **:** Guéneau
Last name

Prénom : Armaël
First name

Sexe : ☐ **F** ☒ **M**
Sex

Nom utilisé pour vos publications (facultatif) :
Name used for your publications (optional):

[1] Il s'agit du nom usuel figurant sur vos pièces d'identité
It is the name appearing on your identity cards

DEPOT DE VOTRE CANDIDATURE
SUBMITTING YOUR APPLICATION

Le dossier de candidature doit comprendre :

- Formulaire 1 : Parcours professionnel
- Formulaire 2 : Description synthétique de l'activité antérieure
- Formulaire 3 : Contributions majeures
- Formulaire 4 : Programme de recherche
- Formulaire 5 : Liste complète des contributions

CRCN & ISFP :

- Les rapports de thèse ou de doctorat (si disponibles)
- Une copie des derniers titres et diplômes
- Une photographie récente de la candidate / du candidat (facultative)

DR2 :

- Les rapports d'habilitation à diriger des recherches (si applicable)
- Une copie des derniers titres et diplômes
- Une photographie récente de la candidate / du candidat (facultative)

The application file must include:

- *Form 1: Professional history*
- *Form 2: Summary of your past activity*
- *Form 3: Major contributions*
- *Form 4: Research program*
- *Form 5: Complete list of contributions*

CRCN & ISFP:

- *PhD dissertation reports (when available)*
- *A copy of most recent titles and diplomas*
- *A recent photography of the applicant (optional)*

DR2:

- *Habilitation dissertation reports (if applicable)*
- *A copy of most recent titles and diplomas*
- *A recent photography of the applicant (optional)*

SOMMAIRE / ***SUMMARY***

Formulaire 1 — PARCOURS PROFESSIONNEL
Form 1 — PROFESSIONAL HISTORY

1) Parcours Professionnel / *Professional history*

Situation professionnelle actuelle / ***Current professional status***

Statut et fonction[2] / *Position and Status[2]*: Post-doc
Etablissement (ville - pays) / *Institution (city - country)*: Aarhus University (Aarhus, Danemark)
Date d'entrée en fonction / *Start*: 1er Janvier 2020
[ ] Sans emploi / *Without employment*

Expériences professionnelles antérieures /***Previous professional experiences***

Date début *Start*	Date fin *End*	Etablissement *Institution*	Fonction et statut[2] *Position and status[2]*
Janvier 2020	—	Université d'Aarhus (Danemark)	Post-doctorant
Septembre 2016	Décembre 2019	Inria Paris, équipe Gallium	Doctorant
Septembre 2016	Mai 2019	Université Paris 7	Mission d'enseignement
Septembre 2012	Juillet 2016	ENS de Lyon	Élève fonctionnaire stagiaire

Nombre d'années d'exercice des métiers de la recherche après la thèse / *Number of years of professional research experience after the PhD:* 1

2) Interruptions de carrière/*Career breaks*

3) Encadrement d'étudiants et de jeunes chercheurs / *Supervision of students and early-stage researchers*

4) Encadrement de développements technologiques (logiciel, matériel, robotique) / *Supervision of technological development (software, hardware, robotics)*

5) Responsabilités collectives / *Responsibilities*

Relecture d'articles de conférence : ICFP 2020, CPP 2021, ESOP 2021
Relecture d'articles de journal : JFP
Membre du **comité d'évaluation d'artefacts** : ICFP 2018
(https://icfp18.sigplan.org/committee/icfp-2018-artifact-evaluation-artifact-evaluation-committee)

2018-2019 : co-organisateur du **séminaire des doctorants** de l'Inria Paris (https://www.rocq.inria.fr/semdoc/, 2 exposés par mois). Il s'agit d'un séminaire organisé par des doctorants, invitant d'autres doctorants à présenter leur travail, à destination de l'ensemble de la communauté scientifique du centre de recherche. Les thèmes abordés sont aussi divers que ceux couverts par les différentes équipes de l'Inria.

6) Management (si pertinent) / *Management (if relevant)*

7) Mobilité (si pertinent) / *Mobility (if relevant)*

Depuis janvier 2020, je suis post-doctorant à l'université d'Aarhus au Danemark, où je m'intéresse à la vérification formelle de propriétés de sécurité pour des machines à capabilités. Je collabore avec Lars Birkedal, Aïna Linn Georges, Amin Timany, Alix Trieu (Université d'Aarhus), Dominique Devriese (Vrije Universiteit, Bruxelles), Frank Piessens et Thomas Van Strydonck (KU Leuven).
Avant ma thèse à l'équipe Gallium de l'Inria Paris, j'y ai effectué deux stages de recherche, en L3 (juin-juillet 2013) et en M2 (mai-juillet 2015), au cours desquels j'ai travaillé avec François Pottier (Inria Paris) et Arthur Charguéraud (Inria et Université de Strasbourg) (par la suite mes encadrants de thèse) et Jonathan Protzenko (alors à l'Inria Paris).

[2] Indiquez avec précision chaque situation statutaire. Par exemple : pour une situation d'agent titulaire de la fonction publique, précisez le corps et le grade de rattachement, pour une situation de salarié(e) du secteur privé ou d'agent non titulaire d'un établissement public, précisez la nature du contrat salarial, etc.

[2] *For each position, indicate grade or rank. For example, for a tenured civil servant position, indicate the branch and rank, for a private sector position or non-tenured position in a public institution, indicate the nature of the work contract, etc.*

J'ai effectué 2 stages dans le cadre de ma 4ème année d'ENS (année scolaire 2015-2016). Durant un premier stage, j'ai travaillé avec KC Sivaramakrishnan dans le groupe OCamllabs de l'université de Cambridge (UK). Pour le second stage, j'ai travaillé à l'université Chalmers (Göteborg, Suède) avec Magnus Myreen, dans le cadre du projet CakeML, ce qui m'a amené à collaborer également avec Ramana Kumar et Michael Norrish (Data61, CSIRO, Australie).
J'ai effectué mon stage de M1 (mai-juillet 2014) à Imperial College (UK), au cours duquel j'ai travaillé avec Jules Villard.

8) Enseignement (si pertinent) / *Teaching (if relevant)*

2016-2019 : mission d'enseignement à l'université Paris 7 – trois services de 64h, soit 192h au total. Enseignement de Java et Python en L1. Travaux dirigés d'introduction à la compilation (L1) et langages formels et automates (L2). Enseignement d'OCaml en L3.
Dans le cadre du cours d'introduction à la programmation fonctionnelle en OCaml, adaptation des sujets papier existants et conception de nouveaux sujets de travaux pratiques pour la plateforme Learn-OCaml, qui permet en particulier d'implémenter un ensemble de tests automatisés afin de guider les étudiants.

9) Diffusion de l'information scientifique (si pertinent) / *Dissemination of scientific knowledge (if relevant)*

2017-2019: Co-rédaction et édition du blog Gagallium (http://gallium.inria.fr/blog) de l'équipe Inria Gallium. 3 articles rédigés sur les 11 articles publiés.

10) Visibilité (si pertinent) / *Visibility (if relevant)*

11) Eléments divers / *Other relevant information*

Vérification formelle de la correction et de la complexité de programmes. Un bon algorithme doit être correct. Pourtant, il est courant que des erreurs se glissent lors de son implémentation concrète. En particulier, qu'un algorithme renvoie le bon résultat n'est pas suffisant : on s'attend également à ce qu'il soit efficace. L'art de l'algorithmique s'intéresse à la conception d'algorithmes complexes mais plus efficaces qu'une solution naïve : leur raison d'être est alors leur bonne *complexité asymptotique*. Pendant ma thèse, j'ai travaillé à la conception d'une méthodologie permettant de vérifier formellement la complexité asymptotique de programmes en même temps que leur correction fonctionnelle.
Un aspect distinctif de l'approche est l'accent mis sur l'utilisation de la notation O due à Landau et de techniques de preuves modulaires et robustes, afin de spécifier et structurer les preuves de la complexité de programmes [3,6]. Une logique de programmes expressive est utilisée, permettant de vérifier des programmes élaborés et des bornes de complexité amortie complexes, pouvant notamment dépendre d'invariants fonctionnels. Concrètement, j'ai développé cette méthodologie sous la forme d'un outil de vérification de programmes OCaml impératifs, embarqué dans l'*assistant de preuve* Coq.
Je me suis également appliqué à en démontrer l'utilisation pratique. En collaboration avec Jacques-Henri Jourdan, j'ai vérifié la correction et la complexité asymptotique amortie d'un algorithme de l'état de l'art, pour la détection incrémentale de cycles dans un graphe, et qui n'a lui-même été publié que récemment (en 2016). Ce travail [2] nous permis de découvrir une amélioration à l'algorithme lui-même. Par ailleurs, notre implémentation vérifiée a été intégrée au logiciel libre `dune`, corrigeant par là plusieurs bogues de complexité et améliorant ses performances.

Propriétés de sécurité haut niveau pour du code bas niveau sur des machines à capabilités. Un grand nombre d'attaques de sécurité reposent sur un attaquant corrompant la mémoire du programme cible (par exemple via un *buffer overflow*), permettant alors de compromettre le flot de contrôle du programme ou d'écraser des données privées. Malheureusement, les architectures matérielles conventionnelles permettent difficilement de se protéger contre ce genre d'attaques. Les machines à *capabilités* sont une forme d'architecture matérielle conçue pour fournir une sécurité accrue, au niveau du matériel, et qui voit récemment un regain d'intérêt notamment grâce aux efforts du projet CHERI, suivi dernièrement par ARM avec la fabrication du prototype industriel Morello.
La ligne de recherche à laquelle je prends part pendant mon post-doc s'intéresse aux moyens de garantir et de raisonner formellement sur les propriétés de sécurité "haut niveau" qu'il est possible de garantir pour des programmes bas niveau sur une machine à capabilités. Avec l'aide de mes collaborateurs, la réalisation principale de mon post-doc a été la conception et la formalisation d'une nouvelle convention d'appel sûre basée sur les capabilités [1]. Ce travail constitue une innovation non seulement sur la manière d'utiliser les capabilités, mais également concernant les techniques de raisonnement : nous définissons un modèle sémantique des garanties fournies par la machine, entièrement mécanisé en Coq grâce à l'utilisation de la logique de programmes Iris. J'ai ensuite mené la rédaction d'un article à visée plus pédagogique [8], illustrant en détails les éléments clefs de notre méthodologie. Finalement, nous nous intéressons à la conception d'une convention d'appel qui soit *fully abstract*, garantissant des propriétés de sécurité supplémentaires (notamment, de type "confidentialité"), et qui motiverait l'introduction d'un nouveau type de capabilités [5].

Logique de séparation pour la vérification de programmes CakeML impératifs avec entrées-sorties. Le compilateur CakeML est un compilateur vérifié formellement pour un très large sous-ensemble de ML, permettant de garantir que la compilation d'un programme source vers le code machine n'introduit pas de nouveau bugs. Je suis l'auteur initial d'un outil [4] permettant de prouver la correction fonctionnelle de programmes CakeML, d'une manière pouvant ensuite se combiner avec le théorème de correction du compilateur CakeML, afin d'obtenir un théorème de correction pour le code machine obtenu après compilation. Ce travail adapte une approche existant en Coq avec l'outil CFML, mais qui n'était pas totalement vérifiée et ne s'appliquait qu'à un langage noyau. Dans ce travail, j'ai formalisé de bout en bout la correction de l'approche, et l'ai étendue en la généralisant à l'ensemble des fonctionnalités du langage CakeML, ajoutant notamment le support pour les exceptions et entrées-sorties.

Itération et transfert de ressources en Mezzo. Le langage de programmation Mezzo fait partie d'une famille de langages expérimentaux avec des systèmes de types expressifs dits "sous-structuraux", ayant inspiré le langage Rust, en plein essor dans l'industrie récemment. Lors d'un stage dans l'équipe Gallium, j'ai utilisé Mezzo afin d'étudier, dans ce cadre, les façons possibles de programmer des itérateurs en exprimant les différents transferts de ressources correspondants [7].

Contributions au compilateur OCaml. Pendant ma thèse, j'ai eu l'occasion de contribuer au développement du compilateur du langage OCaml, et j'en suis actuellement un des 26 mainteneurs. J'ai notamment contribué à nettoyer et restructurer la partie du compilateur dédiée à l'affichage des messages d'erreur, et j'ai contribué à implémenter un certain nombre d'améliorations aux messages d'erreurs qui avaient été proposées par Arthur Charguéraud, et qui nécessitaient une intégration non-triviale avec l'algorithme d'inférence de types.

Formulaire 3 — CONTRIBUTIONS MAJEURES
Form 3 — MAJOR CONTRIBUTIONS

Fiche 1 : Vérification formelle de la correction et complexité de programmes

1. Description de la contribution / *Description of the contribution*

Je suis l'auteur principal d'une méthode pour la preuve formelle de la correction et complexité amortie de programmes, avec bornes de complexité asymptotique en O. Celle-ci est embarquée dans l'assistant de preuve Coq, et permet donc d'établir des preuves rigoureuses et mécanisées de la correction et complexité d'un programme. Son application principale a été la vérification d'un algorithme récent de détection incrémentale de cycles dans un graphe, issu de la recherche en algorithmique et publié récemment par Bender, Fineman, Gilbert et Tarjan en 2016.

2. Contribution personnelle de la candidate ou du candidate / *Personal contribution of the applicant*

Il s'agit en grande majorité d'un travail personnel, fait durant ma thèse en collaboration avec mes encadrants François Pottier et Arthur Charguéraud. J'ai également collaboré avec Jacques-Henri Jourdan lors du travail sur la vérification de l'algorithme de graphe. Jacques-Henri a contribué à l'analyse fine des invariants de complexité de l'algorithme, qui étaient un prérequis à la preuve formelle. Je suis entièrement l'auteur des améliorations faites à la logique de programme, l'implémentation de l'outil de vérification, l'implémentation de l'algorithme de graphe et sa vérification formelle en Coq.

3. Originalité et difficulté / *Originality and difficulty*

La majorité des travaux existants se concentrent essentiellement soit sur des approches automatisées mais limitées dans l'ensemble des programmes et bornes gérées, ou dans des approches embarquées dans un assistant de preuve qui demandent un travail de comptage explicite des instructions effectuées par le programme.
Ce travail propose un moyen d'établir des bornes de complexité potentiellement très complexes, tout en évitant de raisonner à bas niveau en terme de comptes précis d'étapes de calcul élémentaires. Ainsi, il s'appuie sur des spécifications *modulaires* grâce à la notation O, et l'utilisation de techniques de vérification *robustes* vis-à-vis de changements mineurs dans le code du programme, via des mécanismes d'inférence de coût guidés par l'utilisateur, et de manipulation de constantes symboliques pour exprimer des fonctions de coût abstraites.
Il s'agit également du premier travail de vérification de complexité qui démontre son utilité pour s'attaquer à la vérification d'un algorithme de l'état de l'art, plutôt que des algorithmes vieux de quelques décennies et bien connus de la littérature en algorithmique.

4. Validation et impact / *Validation and impact*

L'implémentation de l'algorithme de détection incrémentale de cycle est disponibles sous license *open-source*, en tant qu'une bibliothèque réutilisable. Celle-ci a été en particulier intégrée au logiciel libre `dune`, un gestionnaire de compilation (*build system*) populaire de l'écosystème OCaml. Il s'agit du *build system* pour OCaml le plus utilisé (parmi les 3156 paquets disponibles via opam, 1638 dépendent de `dune`), et est toujours activement développé (depuis l'intégration de mon implémentation vérifiée en Mai 2019, 24 nouvelles versions de `dune` ont aujourd'hui été publiées).
`Dune` utilisait initialement une implémentation non vérifiée du même algorithme de détection incrémentale de cycles. J'ai fait le travail permettant d'intégrer mon implémentation vérifiée au sein de `dune`, contribution qui a été acceptée par ses mainteneurs et en fait maintenant partie. L'implémentation non-vérifiée utilisée précédemment comportait plusieurs bogues de complexité ; j'ai alors pu mesurer ma nouvelle implémentation être jusqu'à 7 fois plus rapide sur un scénario concret.

5. Diffusion / *Dissemination*

Ma thèse porte sur ces travaux, ainsi que les articles ESOP'18 [3] et ITP'19 [2]. Un aspect du travail sur les mécanismes de preuves formelle a été présenté au workshop Coq [6]. L'outil de vérification que j'ai développé est disponible en ligne sous license *open-source* [9], ainsi qui l'implémentation de l'algorithme de graphe et sa preuve formelle [10].
Mon travail m'a mis en contact avec Maximilian Haslbeck, doctorant avec Tobias Nipkow à TU Munich qui travaille sur des sujets similaires, et m'a valu de visiter le groupe "Logic and Verification" pendant une semaine pour y présenter et discuter mes travaux.
J'ai écrit quatre billets de blog (à destination de la communauté scientifique) : un, deux, trois, quatre.
(`http://gallium.inria.fr/blog/formally-verified-complexity-with-cfml-part-{1,2,3}`, `http://gallium.inria.fr/blog/incremental-cycle-detection`)

Fiche 2 : Propriétés de sécurité haut niveau pour machines à capabilités

1. Description de la contribution / *Description of the contribution*

Les architectures matérielles d'aujourd'hui fournissent peu de moyens de se protéger contre des attaques de sécurité où un attaquant corrompt la mémoire d'un programme vulnérable (par exemple via un *buffer overflow*) ; un CPU traditionnel ne fournit en effet que des mécanismes de cloisonnement à gros grain, typiquement utilisés par le système d'exploitation à l'échelle d'un processus entier. Les machines à capabilités (*capability machines*) sont une forme d'architecture matérielle fournissant des mécanismes de sécurité à grain fin. Une machine à capabilités associe des méta-données à chaque mot machine (stocké dans les registres ou la mémoire), distinguant tout d'abord entre entiers et capabilités, les capabilités jouant alors le rôle de pointeurs permettant d'accéder à la mémoire. Les méta-données associées à chaque capabilité restreignent son accès à une certaine région mémoire, pour une certaine permission ; la machine garantit l'intégrité de ces méta-données, qui ne peuvent être manipulées que selon des règles bien précises.
Éliminer les buffer overflows n'est alors qu'un des cas d'application possibles : les capabilités tirent leur force des différentes manières dont elles peuvent être combinées, permettant alors de garantir avec assurance des propriétés de sécurité complexes. Il se pose alors, d'une part, la question de quelle utilisation concrète des capabilités faire pour garantir telle ou telle propriété ; et d'autre part, de comment raisonner sur les garanties formelles obtenues, pour s'assurer qu'elles correspondent à ce que l'on espère.
La contribution principale de ce travail est le développement de modèles sémantiques expressifs, permettant de raisonner à haut niveau sur les propriétés garanties à bas niveau par une machine à capabilités ; le tout étant formalisé en Coq grâce à l'utilisation de la logique de programmes Iris. Les applications ont alors été la vérification d'une nouvelle convention d'appel sûre [1], mettant en jeu un usage particulièrement subtil des capabilités, puis la déclinaison de la même approche dans un cadre plus simple à but pédagogique [2]. Ce travail est également une opportunité pour considérer l'introduction de nouvelles capabilités aux machines futures, motivées par la volonté de préserver des propriétés encore plus fortes [3].

2. Contribution personnelle de la candidate / du candidat / *Personal contribution of the candidate*

Ce travail est un effort collectif, réalisé en collaboration avec Lars Birkedal, Dominique Devriese, Aïna Linn Georges, Sander Huyghebaert, Thomas Van Strydonck, Amin Timany et Alix Trieu. J'ai rejoint le projet au cours de son développement, mais je fais maintenant partie des collaborateurs principaux.
Lors du travail sur la convention d'appel sûre, j'ai participé à l'élaboration du modèle au cœur de la relation logique, et j'ai ensuite surtout contribué à la vérification concrète des programmes utilisés pour illustrer les propriétés d'encapsulation de l'état local et de flot de contrôle bien parenthésé.
Je suis l'initiateur et l'auteur principal de l'article JFLA'21 [8]. Je me suis chargé de la rédaction de l'article et la vérification de l'étude de cas principale; et Aïna a porté le développement Coq issu de l'article POPL publié précédemment au cadre plus simple envisagé pour l'article JFLA.

3. Originalité et difficulté / *Originality and difficulty*

Il existe aujourd'hui relativement peu de travaux s'attaquant à la vérification de propriétés de sécurités pour les machines à capabilités. Le regain d'intérêt au sujet de ces machines au cours des dix dernières années est majoritairement dû aux efforts du groupe CHERI (Cambridge, UK), qui s'est majoritairement consacré à démontrer la viabilité de machines à capabilités réalistes, à définir formellement la sémantique de telles machines, et, plus récemment, à établir des propriétés de "monotonicité des capabilités" relativement syntaxiques.
L'originalité de ce travail est de suivre une approche sémantique, permettant de raisonner explicitement sur le comportement d'un programme connu interagissant avec un adversaire inconnu, les deux s'exécutant sur la même machine. Cette approche se place dans la continuation du travail effectué par Lau Skorstengaard, qui suivait des objectifs similaires mais utilisait un modèle logique et des preuves développées sur papier : notre modèle est lui entièrement formalisé en Coq, et se place à un niveau d'abstraction plus élevé, grâce à l'utilisation de la logique Iris.

4. Validation et impact / *Validation and impact*

Ce travail a un fort potentiel d'impact. Le constructeur ARM a récemment lancé la fabrication d'un prototype de machine à capabilités CHERI (https://www.morello-project.org), permettant d'évaluer en pratique les propositions diverses d'utilisations de capabilités qui ont pu être faites. Ce travail nous permet donc de renseigner à la fois la conception des futurs systèmes logiciels utilisant les capabilités, et de fournir des arguments pour l'introduction de nouvelles capabilités (comme les capabilités *uninitialized* utilisées dans l'article POPL'21). Nous sommes d'ailleurs en contact régulier avec le groupe de Peter Sewell, qui fait partie du projet CHERI, et collabore avec l'équipe de développement chez ARM.

5. Diffusion / *Dissemination*

Publications : le travail sur la convention d'appel a été publié à POPL'21 [1], et l'article pédagogique illustrant notre méthodologie est à apparaître aux JFLA'21 [8]. Une présentation au workshop PriSC [5] fait l'état de travaux préliminaires pour la conception d'une convention d'appel *fully abstract*. Les développements Coq sont accessibles en ligne [11].

Fiche 3 : Logique de séparation pour la vérification de programmes CakeML impératifs avec entrées-sorties

1. Description de la contribution / *Description of the contribution*

Le compilateur CakeML est un compilateur pour une très large fraction du langage ML, muni d'une preuve de correction mécanisée en HOL4 démontrant mathématiquement la correction du code machine généré. Il s'agit d'un projet très actif (avec 33 publications reliées, depuis 2012), et dont les performances du code généré sont comparables à des compilateurs ML de l'état de l'art.
Je suis l'auteur principal d'un outil permettant de prouver la correction fonctionnelle de programmes CakeML, d'une manière pouvant se combiner avec avec le théorème de correction du compilateur, pour obtenir un résultat de correction à propos du code machine obtenu.
En Coq, l'outil CFML d'Arthur Charguéraud fournissait un mécanisme de preuve de programmes ML impératifs en logique de séparation, mais n'était pas vérifié de bout en bout (produisait des axiomes), et était limité à un langage noyau. Lors de ce travail, sur l'échelle d'un stage de 5 mois, j'ai adapté une approche s'appuyant sur la génération de *formules caractéristiques* en la portant en HOL4, en l'intégrant à l'écosystème CakeML, et en l'améliorant de deux manières essentielles : d'une part, en formalisant de bout en bout la méthodologie, prouvant ainsi une fois pour toutes que les formules caractéristiques générées par l'outil sont correctes (au lieu de devoir les admettre en axiome); et d'autre part, en la généralisant à un langage plus riche, en ajoutant notamment le support pour les exceptions et entrées-sorties.

2. Contribution personnelle de la candidate / du candidat / *Personal contribution of the candidate*

Il s'agit d'un travail dans lequel j'ai été aidé et supervisé par Magnus Myreen, Ramana Kumar et Michael Norrish, mais le développement de l'outil de preuve et la vérification formelle de sa correction est en grande majorité de mon fait. L'extension concernant la vérification des entrées-sorties est majoritairement due à Magnus, et Michael a développé le programme `cat` vérifié servant d'étude de cas dans notre article.

3. Originalité et difficulté / *Originality and difficulty*

Historiquement, le compilateur CakeML disposait d'un mécanisme, implémenté en HOL4, permettant d'obtenir des programmes CakeML vérifiés à partir de fonctions de la logique. Ce schéma de traduction ne permet toutefois que de produire des programmes CakeML purement fonctionnels, ce qui a motivé ce travail, afin de pouvoir vérifier la correction de programmes fonctionnels et impératifs tirant parti de l'ensemble des fonctionnalités du langage CakeML. L'outil CFML d'Arthur Charguéraud mentionné précédemment ne supportait qu'un langage moins réaliste, et n'était pas vérifié de bout en bout.
Plus généralement, on peut citer un nombre varié de travaux s'attachant à établir la correction formelle de programmes et à transporter ce résultat après compilation ; par exemple VST (the Verified Software Toolchain) pour la vérification de programmes C en Coq, ou l'approche de vérification par raffinement de Peter Lammich en Isabelle/HOL.
La force de ce travail est son intégration avec le très actif "écosystème CakeML". D'une part, on peut ainsi obtenir du code binaire vérifié aux performances compétitives ; d'autre part, cet outil de vérification est utile et utilisé pour le développement du compilateur CakeML lui-même, et permet donc de supporter la recherche autour de CakeML.

4. Validation et impact / *Validation and impact*

Ce travail a été intégré au développement du compilateur CakeML. Il a été par la suite utilisé pour la vérification de la bibliothèque standard de CakeML et d'applications (Férée et al, VSTTE'18), et par ailleurs étendu ou utilisé d'une manière ou d'une autre dans plusieurs travaux ultérieurs (Ho et al. à IJCAR'18, Bohrer et al. à PLDI'18, et Pohjola et al. à ITP'19).
Il a également motivé le développement par Arthur Charguéraud d'une refonte de CFML intitulée CFML2, et cette fois vérifiée de bout en bout, dont les développements ont fait l'objet d'un article à ICFP 2020.

5. Diffusion / *Dissemination*

Ce travail a été publié à ESOP'17 [4]. Le code appartient à la distribution officielle de CakeML (https://github.com/cakeml/cakeml), accessible sous license open-source.

Formulaire 4 — PROGRAMME DE RECHERCHE
Form 4 — RESEARCH PROGRAM

☒ Je souhaite candidater dans l'équipe-projet, ou les équipes-projets suivante(s) : TOCCATA / CELTIQUE

Intitulé du programme de recherche : **Preuves de programmes en contexte**

La recherche en preuves formelles de ces 10 dernières années a vu le succès de projets ambitieux tels CompCert, CakeML ou seL4, qui établissent de bout en bout la correction de logiciels complets, compilateurs ou systèmes d'exploitation. Vérifier formellement un logiciel de cette ampleur requiert des efforts considérables, plusieurs dizaines d'années-homme. Pourtant, ces efforts ne permettent finalement de vérifier que quelques dizaines de milliers de lignes de code, là où l'implémentation d'un logiciel moderne – par exemple celui embarqué dans une voiture récente – atteint les 100 millions de lignes.
Il est alors difficile d'envisager tout vérifier. Heureusement, seule une fraction du logiciel en question est effectivement critique à la sécurité des passagers du véhicule. On espère donc pouvoir intégrer dans une même voiture des *composants* logiciels critiques (vérifiés, sécurisés) avec des composants qui n'en sont pas et que l'on ne prend pas la peine de vérifier. Il est alors crucial de disposer de garanties très fortes concernant l'intégration entre ces composants et leurs interactions possibles. Comment vérifier formellement la correction de composants logiciels placés dans le contexte d'un système entier non vérifié ?

Preuves de sécurité et machines à capabilités
Une machine à capabilités est un type de microprocesseur étendant le fonctionnement d'une architecture conventionnelle avec des mécanismes de protection mémoire et de séparation des privilèges, garantis directement au niveau du matériel ("*security by design*"). Leur utilisation permet d'éliminer à la source une large classe de failles de sécurité, dont notamment celles causées par les célèbres *buffer overflows*. Plus généralement, ces machines fournissent un terrain fertile où étudier de nouvelles manières de concevoir des systèmes logiciels plus sûrs, permettant un cloisonnement à grain plus fin, ou encore avec une base de confiance plus réduite.
Les machines à capabilités sont aujourd'hui un candidat crédible pour devenir l'architecture de demain, envahissant nos téléphones et ordinateurs, mais n'ont reçu pour l'instant qu'une attention limitée de la part de la communauté de recherche en langages de programmation – à l'exception du projet CHERI (Cambridge, UK) qui se consacre depuis une décennie à la conception de machines à capabilités réalistes et d'une pile logicielle associée.
Dans ce contexte, l'utilisation de méthodes formelles est pourtant cruciale : les capabilités ne fournissent une sûreté accrue qu'à condition de bien les utiliser ! Concevoir des scénarios complexes d'utilisation des capabilités ou proposer l'introduction de nouvelles capabilités n'est pas une tâche aisée : les capabilités opèrent à bas niveau, et la sécurité qu'elles fournissent dépend de leur interaction avec l'ensemble des fonctionnalités de la machine. Il est donc important de pouvoir raisonner formellement sur les garanties fournies par la machine, dans des scénarios concrets ; sans cadre formel, il est aisé de commettre des erreurs potentiellement critiques.
Lors de mon post-doc, j'ai pris part à une ligne de recherche s'appuyant sur des outils formels pour raisonner sur une convention d'appel sûre utilisant les capabilités. J'espère étendre et redéployer cette méthodologie pour établir formellement la sûreté d'autres composants système, qui seraient utiles pour bâtir une pile logicielle vérifiée utilisant les capabilités, et ce dans un contexte où un composant peut interagir avec du code inconnu sans la coopération du système d'exploitation.
C'est un projet qui, pour être mené à bien, nécessite d'une part une bonne compréhension des enjeux d'un point de vue système, afin de garantir la pertinence du code et des propriétés vérifiées; d'autre part une maîtrise de techniques formelles avancées (relations logiques et la logique de programmes Iris); et finalement des compétences pratiques de preuve de programmes pour pouvoir mener à bout la vérification de programmes bas niveau complexes.
Un exemple d'un tel composant système est un allocateur mémoire, pointé du doigt par le rapport du Microsoft Security Response Center sur CHERI comme étant un élément particulièrement sensible d'un système à capabilités. Le groupe CHERI s'intéresse à la conception d'allocateurs mémoire garantissant une forme de "sûreté temporelle" éliminant les attaques du type "*use after free*" (CHERIvoke (IEEE Micro 2019) puis Cornucopia (IEEE S&P 2020)).
Ces travaux se concentrent sur l'évaluation pratique de mécanismes garantissant cette propriété. D'un point de vue formel, tout reste à faire : d'une part arriver à énoncer la propriété de sécurité que l'on attend d'un tel allocateur mémoire (sachant qu'en pratique on s'intéresse à une notion relachée de sûreté temporelle, pour des raisons d'efficacité), et ensuite effectivement vérifier cette propriété vis-à-vis d'une implémentation concrète.
Si mon travail de post-doc s'est principalement intéressé à raisonner à propos de code machine, il est légitime de vouloir profiter de la sûreté fournie par une machine à capabilités tout en écrivant le code dans un langage de plus haut niveau. À plus long terme, je compte donc m'intéresser à la vérification de schémas de compilation sûrs ciblant des machines à capabilités. Un premier candidat serait le langage CHERI-C, actuellement développé par le groupe CHERI comme une extension de C, et pour lequel il s'agirait tout d'abord de fournir une spécification formelle. Un autre candidat serait le langage CakeML (et son compilateur vérifié) : l'utilisation de capabilités permettrait alors de sécuriser son mécanisme

d'interaction avec du code externe (FFI, "*foreign function interface*"). Dans les deux cas, une convention d'appel sûre comme étudiée dans mon article POPL'21 serait un élément essentiel.

Bonne interopérabilité via le typage et composants multi-langages

Les bibliothèques d'un langage de haut niveau sont une autre forme de composants logiciels réutilisables. Ceux-ci sont composés au sein d'un même langage, qui fournit généralement des mécanismes d'encapsulation bénéfiques à la programmation modulaire (qui peuvent être statiques ou dynamiques). Comment s'appuyer sur ces mécanismes d'encapsulation pour vérifier que les garanties établies pour une bibliothèque donnée seront préservées lors de son utilisation dans un contexte non vérifié ? En particulier, je m'intéresse à relier des spécifications établies en logique de séparation (qui raisonnent donc en terme de ressources et de possession unique) avec des interfaces exprimées dans un système de type à la ML (qui ne permet pas de raisonner sur des ressources ou des questions de possession).

Il s'agit alors de concevoir l'interface de la bibliothèque de façon à pouvoir montrer que les préconditions attendues par le composant vérifié sont garanties par les mécanismes d'encapsulation du langage.

Interaction entre composant vérifié et contexte non vérifié mais bien typé

Dans le cas d'un langage statiquement typé, si l'on vérifie formellement une bibliothèque, puis qu'on l'expose à un contexte bien typé mais non vérifié, le danger est que ce contexte utilise la bibliothèque dans des cas non couverts par la spécification. On a alors "mal" utilisé le composant, et la preuve formelle de sa correction ne s'applique plus !

J'espère montrer que, pour un système de types fournissant une notion d'encapsulation suffisante (comme le système de modules d'OCaml fournissant des types abstraits), alors il est souvent possible de garantir que des spécifications en logique de séparation sont préservées par un contexte typé, soit statiquement, soit en signalant explicitement via une erreur à l'exécution quand ce n'est pas le cas.

Le résultat serait alors une méthode de preuve permettant, pour une bibliothèque munie d'une spécification riche en logique de séparation, de montrer que l'on peut spécifier son interface en n'utilisant seulement les types du langage (mais possiblement après l'avoir équipée de tests dynamiques légers). Alors, on obtiendrait que pour tout contexte arbitraire bien typé, si l'exécution de ce contexte utilisant la bibliothèque ne conduit pas à l'échec d'un test dynamique, alors ce contexte utilise en fait la bibliothèque correctement selon sa spécification formelle.

Composants multi-langages : interface haut niveau et implémentation bas niveau

En pratique, un nombre non négligeable de bibliothèques des langages de haut niveau sont partiellement implémentées avec du code bas niveau (typiquement écrit en C), pour interopérer avec des bibliothèques système ou pour des raisons d'efficacité. Les détails de la FFI (qui définit comment lier du code de haut niveau avec du code plus bas niveau) varient d'un langage à un autre. Dans le cas d'une FFI haute performance, comme celle offerte par OCaml, le programmeur a un contrôle fin lui permettant d'implémenter une bibliothèque la plus efficace possible; mais en retour, il doit obéir à de nombreuses restrictions lors de son interaction avec le langage de haut niveau. Malheureusement, après 25 ans d'existence, OCaml n'est toujours pas muni d'une spécification ou d'un modèle formel de l'interface FFI exposée au programmeur. La programmation de code d'interfaçage entre OCaml et C reste un art réservé aux initiés, et le débogage est difficile.

J'aimerais pouvoir décrire formellement l'ensemble des mécanismes nécessaires pour vérifier de bout en bout un composant multi-langage. Comment vérifier formellement qu'un composant, implémenté dans un langage bas niveau et exposé selon une interface haut niveau bien typée : 1) est correct, 2) satisfait son interface, et 3) que cette interface ne permet pas de briser les bonnes propriétés du langage environnant ?

Une des difficultés provient des différentes stratégies de gestion mémoire : la plupart des langages de haut niveau implémentent une gestion automatique de la mémoire via un GC (*garbage collector*), là où le langage C permet une gestion manuelle de la mémoire. Il devient alors nécessaire de spécifier formellement l'interface offerte par le GC au code bas niveau. Par exemple, si le GC peut déplacer en mémoire les valeurs du langage de haut niveau, partager un pointeur vers une valeur n'est possible que sous certaines conditions. Je compte m'attaquer à ce problème particulier en spécifiant, en logique de séparation, l'interface bas niveau offerte par un GC, en fonction des différentes fonctionnalités offertes par le *runtime* de différents langages (OCaml, .NET, Go, Lua, ...); et ensuite démontrer l'application de cette spécification formelle pour vérifier la correction d'exemples concrets de code C interagissant avec du code OCaml via la FFI d'OCaml.

Vérification de correction et complexité asymptotique

Une certaine catégorie de bibliothèques hautement réutilisables sont celles implémentant des structures de données et algorithmes, et dont l'implémentation est généralement subtile. La spécification mathématique de telles bibliothèques inclut alors non seulement leur correction mais également leur garantie de complexité : la raison d'être d'un algorithme subtil est généralement sa complexité, par opposition à une solution naïve. Un bug de complexité dans l'implémentation d'une bibliothèque peut de plus exposer son utilisateur à des attaques de type "déni de service", où un attaquant épuise les ressources de la machine (temps ou mémoire).

J'ai développé dans ma thèse une méthode de preuve pour la correction et la complexité, utilisant la notion de *crédits temps* en logique de séparation. Toutefois, cette approche (embarquée dans Coq comme une extension de CFML) est pour l'instant relativement difficile d'utilisation et reste réservée aux experts.

Intégration avec des outils automatiques d'inférence de complexité

Je propose de travailler à l'intégration de l'approche développée pendant ma thèse – difficile d'emploi mais très expressive – avec des outils existants *d'inférence* de complexité, qui sont complètement automatiques mais plus limités. La majorité des travaux existants se concentrent en effet essentiellement soit sur des approches automatisées (limitées dans la gamme de programmes et bornes supportées) ou sur des approches embarquées dans un assistant de preuve et requérant une vérification manuelle.
J'aimerais pouvoir utiliser un des outils automatisés comme "procédure auxiliaire" (de décision et d'inférence) depuis un assistant de preuve pour les morceaux de programme rentrant dans le cadre de l'outil, et en utilisant une approche plus manuelle pour les bornes ou les programmes plus complexes.
À cette fin, les travaux de Quentin Carbonneaux et de Jan Hoffmann, qui ont développé des outils d'inférence de complexité – respectivement pour du code objet type LLVM et un sous-ensemble de OCaml – sont particulièrement prometteurs. Un premier objectif serait alors de traduire un certificat issu de l'outil de Quentin Carbonneaux ou une dérivation de type issue de RAML (l'outil de Jan Hoffmann) en une dérivation de preuve en logique de séparation avec crédits temps, utilisable en Coq. Plus difficile : comment permettre à un outil d'inférence automatique d'utiliser une spécification de complexité établie manuellement ? Un travail théorique supplémentaire est probablement nécessaire pour étendre les outils d'inférence afin de leur permettre de tirer parti de telles spécifications.

Complexité en espace
En plus de bornes de complexité en temps, il est également utile d'établir des bornes de complexité en espace. On pense en particulier au cas de structures de données avec partage (notamment structures de données fonctionnelles ou (semi-)persistantes), dont il est difficile d'analyser la consommation mémoire.
En collaboration avec Jacques-Henri Jourdan, François Pottier et Arthur Charguéraud, je compte m'intéresser à la généralisation des crédits temps en des *crédits mémoire*. La possession de crédits mémoire garantirait alors qu'un certain nombre de cases mémoires sont disponibles ; si l'on prouve une spécification en logique de séparation avec crédits mémoire, alors le nombre de crédits en précondition donnerait une borne sur la quantité maximale de mémoire requise lors de l'exécution.
J'aimerais en particulier m'intéresser au cas d'un langage avec gestion automatique de la mémoire (via un GC), qui est plus subtil et reste à explorer. Techniquement, une difficulté est que l'on aimerait étendre la logique de séparation pour pouvoir raisonner sur la *vivacité* d'une structure en mémoire. Si on libère la mémoire correspondant à une structure (qui était précédemment utilisée), alors on espère obtenir des crédits mémoire en échange. Dans un langage avec gestion automatique de la mémoire, ceci n'est possible que si on est capable de justifier que le reste du programme ne maintient pas la structure vivante. Peut-on définir, au niveau de la logique de séparation, une notion permettant de raisonner de façon modulaire sur des "références uniques" à une structure en mémoire ?

Intégration à Celtique
L'équipe Celtique a une expertise impressionnante à la fois en sécurité et en compilation vérifiée. Un certain nombre de travaux sont par ailleurs menés à bien dans le cadre ambitieux du compilateur vérifié CompCert, incluant par exemple une variante de CompCert préservant les propriétés *constant time* (important pour la sécurité de code cryptographique), ou encore pour garantir des propriétés de cloisonnement (*software fault isolation*) — témoin de la la forte expertise locale en l'assistant de preuve Coq. Il s'agirait donc d'un endroit idéal pour développer les axes de mon projet liés à la sécurité et aux machines à capabilités. Notamment, l'axe concernant la compilation vers des capabilités pourrait profiter des compétences fortes de l'équipe sur le compilateur vérifié CompCert, et pourrait mener à des collaborations directement liées à l'utilisation de CompCert pour cibler des machines à capabilités.
Sandrine Blazy s'intéresse à raisonner formellement à propos de code machine, et plus récemment à raisonner sur l'interaction entre code bas niveau et langage haut niveau dans le cadre d'un compilateur *just-in-time* vérifié. Ce travail soulève un certain nombre de problématiques partagées avec l'axe 2 de mon projet concernant la vérification de composants multi-langages : une collaboration sur le sujet ne pourrait être que bénéfique.
Alan Schmitt et Thomas Jensen sont parmi les concepteurs des "sémantiques squelettiques", qui permettent de raisonner à propos de langages dont la sémantique, si écrite naïvement, serait beaucoup trop grosse pour être humainement utilisable. On retrouve exactement la même problématique à propos des modèles formels de machines à capabilités CHERI réalistes. Celles-ci comportent tellement de détails et de cas d'erreurs différents, qu'on ne sait actuellement pas les intégrer de façon raisonnable avec une logique de séparation comme Iris, d'une manière qui permettrait de prouver la correction de programmes même modestes. Il serait donc très intéressant d'explorer l'utilisation de sémantiques squelettiques, afin de passer à l'échelle et pouvoir raisonner formellement sur des machines à capabilités réalistes.
En plus de son expertise en sécurité et compilation vérifiée, Frédéric Besson est un expert de l'implémentation de procédures vérifiées dans Coq, grâce à son travail sur la bibliothèque Micromega. Son expertise serait très utile pour le développement de procédures vérifiées d'aide à la preuve de complexité, dans le cadre de l'axe 3 de mon projet de recherche.
J'apporterais à l'équipe mon expérience dans l'utilisation de logiques de programmes (notamment logique de séparation, grâce à mon expérience de thèse et de post-doc) pour la vérification de programmes, ainsi que la connaissance des problématiques spécifiques liées aux machines à capabilités — thèmes qui me semblent bien complémenter les compétences de l'équipe en sécurité et vérification.

Formulaire 5 — LISTE COMPLÈTE DES CONTRIBUTIONS[1]

Form 5 — COMPLETE LIST OF CONTRIBUTIONS[1]

1. Publications caractéristiques/*Representative publications*

"Efficient and provable local capability revocation using uninitialized capabilities".
Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Dans *Proceedings of the ACM on Programming Languages, Volume 5,* ***POPL***. Article 6, pages 1–30. Janvier 2021.

"A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification".
Armaël Guéneau, Arthur Charguéraud, François Pottier. Dans *Programming Languages and Systems. European Symposium on Programming (**ESOP**)*, 2018. *Lecture Notes in Computer Science*, volume 10801, pages 533–560. Springer, Cham.

"Verified Characteristic Formulae for CakeML".
Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, Michael Norrish. Dans *Programming Languages and Systems. European Symposium on Programming (**ESOP**)*, 2017. *Lecture Notes in Computer Science*, volume 10201, pages 584–610. Springer, Berlin, Heidelberg.

2. Publications

L'usage de mon domaine a été d'ordonner les auteurs par contribution, en utilisant ensuite l'ordre alphabétique en cas de contributions comparables.

2.1 Revues internationales/*International journals*

[1] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal : "Efficient and provable local capability revocation using uninitialized capabilities". Dans *Proceedings of the ACM on Programming Languages, Volume 5,* ***POPL***. Article 6, pages 1–30. Janvier 2021.

2.2 Conférence internationales avec comité de lecture/*Reviewed international conferences*

[2] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, François Pottier : "Formal Proof and Analysis of an Incremental Cycle Detection Algorithm". Dans *10th International Conference on Interactive Theorem Proving (**ITP**). Leibniz International Proceedings in Informatics (LIPIcs)*, volume 141, pages 18:1–18:20. 2019.

[3] Armaël Guéneau, Arthur Charguéraud, François Pottier : "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification". Dans *Programming Languages and Systems. European Symposium on Programming (**ESOP**)*, 2018. *Lecture Notes in Computer Science*, volume 10801, pages 533–560. Springer, Cham.

[4] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, Michael Norrish : "Verified Characteristic Formulae for CakeML". Dans *Programming Languages and Systems. European Symposium on Programming (**ESOP**)*, 2017. *Lecture Notes in Computer Science*, volume 10201, 584-610. Springer, Berlin, Heidelberg.

2.3 Livres et chapitres de livre/*Books and book chapters*

2.4 Autres publications internationales (posters, articles courts)/*Other international publications (posters, short papers)*

[5] Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Lars Birkedal : "Towards Complete Stack Safety for Capability Machines". Dans *Principles of Secure Compilation (PriSC)*. 2021.

[6] Armaël Guéneau : "Procrastination, a proof engineering technique". Dans *Coq Workshop*. 2018.

[1] Les publications et réalisations les plus significatives devront, dans la mesure du possible, être consultables sur la page web de la candidate ou du candidat.
Most relevant contributions (publications, software) should be, as much as possible, available for consultation via the web page of the applicant.

[7] Armaël Guéneau, François Pottier, Jonathan Protzenko : "The ins and outs of iteration in Mezzo". Dans *Higher-Order Programming with Effects (HOPE)*. 2013.

2.5 Revues nationales/*National journals*

2.6 Conférence nationales avec comité de lecture/*Reviewed national conferences*

[8] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, Lars Birkedal : "Cap' ou pas cap' ? Preuve de programmes pour une machine à capacités en presence de code inconnu". À paraître dans *Journées Francophones du Logiciel Applicatif (JFLA)*. 2021.

2.7 Rapports de recherche et articles soumis/*Research reports and publications under review*

3. Développements technologiques : logiciel ou autre réalisation / *Technology development : software or other realization*

[9] La bibliothèque Coq `big0` implémente les composants centraux de l'approche développée pendant ma thèse. En combinaison avec les bibliothèques Coq [14], [15] et [16], elle permet à un utilisateur de développer une preuve mécanisée, dans l'assistant de preuve Coq, de la correction fonctionnelle et de la complexité asymptotique d'un programme OCaml donné. La bibliothèque contient une formalisation de la notation asymptotique O ainsi que ses propriétés utiles dans le cadre de la vérification de programmes, donne un cadre pour spécifier la complexité de programmes en terme de O, et implémente un mécanisme de synthèse d'expressions de coût, dirigé par l'utilisateur, et qui est au centre du processus de vérification de la complexité d'un programme.

Cette bibliothèque est aujourd'hui toujours à un stade expérimental, mais a été utilisée avec succès pour la vérification formelle d'un certain nombre d'exemples, dont l'algorithme de détection incrémentale de cycles dans un graphe [2,10]. J'en suis le principal auteur et développeur.

`Family=research; Audience=partners; Evolution=basic; Duration=3; Contribution=leader,devel`
11 kLOC (total)

https://gitlab.inria.fr/agueneau/coq-big0

[10] La bibliothèque OCaml `incremental-cycles` implémente un algorithme efficace et formellement vérifié pour la détection incrémentale de cycles dans un graphe (l'algorithme étant dû à Bender, Fineman, Gilbert et Tarjan). L'implémentation en OCaml est accompagnée d'une preuve formelle en Coq de sa correction formelle et de sa complexité asymptotique, démontrant l'utilisation des bibliothèques `big0` [9] et `procrastination` [15]. Le code est réutilisable et peut être installé comme une bibliothèque OCaml standard, fournissant ainsi une implémentation de confiance pour un algorithme subtil de l'état de l'art.

Je suis l'auteur et développeur principal du code et des preuves (à 95%).

`Family=research; Audience=community; Evolution=lts; Duration=1; Contribution=leader,devel`
7 kLOC (total)

https://gitlab.inria.fr/agueneau/incremental-cycles

[11] Les développements Coq `cerise` et `cerise-stack` modélisent formellement des machines à capabilités de type CHERI, et déploient une logique de programmes ainsi qu'une relation logique permettant de raisonner sur des propriétés de sécurité pour des programmes s'exécutant sur une telle machine, et intéragissant avec du code inconnu. Il s'agit plus généralement d'un terrain d'expérimentation supportant la recherche du groupe à Aarhus et à Leuven autour des machines à capabilités.

Le développement `cerise-stack` se consacre à l'étude d'une convention d'appel sûre basée sur la pile et la combinaison de capabilités locales et *uninitialized*, qui fait l'objet de l'article POPL'21 [1]. Le développement `cerise` se place dans un cadre plus simple, s'intéressant à l'utilisation seule de capabilités "objet" pour garantir des propriétés d'encapsulation de l'état locale, comme décrit dans l'article JFLA'21 [8].

Bien que n'étant pas un auteur initial du projet, je fais maintenant partie des développeurs principaux. Je suis actuellement l'auteur d'environ 15% du code source.

`Family=research; Audience=team; Evolution=basic; Duration=1; Contribution=devel`
30 kLOC (total) (cerise), **35 kLOC (total)** (cerise-stack)

https://github.com/logsem/cerise
https://github.com/logsem/cerise-stack

[12] CakeML est un compilateur formellement vérifié en HOL4 pour un sous-ensemble conséquent du langage ML, ciblant plusieurs architectures matérielles, et implémentant de nombreuses optimisations. La preuve formelle de son théorème de correction garantit que le résultat de la compilation se comporte conformément au comportement du programme source.

J'ai contribué un outil permettant la vérification interactive de programmes CakeML, et intégré avec le reste du compilateur et de sa preuve. Cette contribution compte pour environ 15 kLOC.

`Family=research; Audience=community; Evolution=lts; Duration=1; Contribution=devel`
430 kLOC (total)

https://github.com/cakeml/cakeml

[13] OCaml est un langage de programmation fonctionnelle, originaire de l'Inria et activement développé depuis 25 ans, et utilisé dans la recherche (Coq est par exemple implémenté en OCaml), l'éducation (par exemple l'enseignement de la programmation fonctionnelle en classes préparatoires et à l'université) et l'industrie (Janestreet, Facebook, Nomadic Labs, ...).

J'ai contribué au développement du compilateur OCaml, dans les parties liées aux messages d'erreurs de types, et plus généralement concernant l'affichage et la localisation des messages d'erreurs. Ces contributions totalisent quelques milliers de lignes de changements, et interagissent avec le moteur d'inférence de types, composant délicat du compilateur.

`Family=research; Audience=community; Evolution=lts; Duration=2; Contribution=devel`
370 kLOC (total)

https://github.com/ocaml/ocaml

[14] CFML est un outil permettant la vérification formelle de programmes OCaml en Coq, en utilisant la logique de séparation. Il est principalement développé par Arthur Charguéraud, a été utilisé pour la vérification d'un certain nombre de structures de données et d'algorithmes non triviaux, et inclut du support basique pour la logique de séparation avec crédits temps, sur lequel se base ma bibliothèque `big0` [9].

J'ai contribué à CFML en ajoutant le support pour les crédits temps *possiblement négatifs*, extension de la logique de séparation avec crédits temps que j'ai introduite pendant ma thèse, et qui fournit des principes de raisonnement plus agréables. J'ai également aidé à la maintenance du code.

`Family=research; Audience=community; Evolution=basic; Duration=2; Contribution=devel,softcont`
80 kLOC (total)

https://gitlab.inria.fr/charguer/cfml

[15] La bibliothèque Coq `procrastination` fournit du support (sous la forme de *tactiques* Coq) pour une méthode de preuve permettant de raisonner "en arrière" à propos de constantes encore inconnues. Ceci permet, lors d'une preuve intéractive, de supposer l'existence de constantes, puis de collecter lors de la preuve des conditions que ces constantes doivent satisfaire, pour enfin les instancier à une valeur satisfaisant ces conditions.

Il s'agit d'une petite bibliothèque, mais qui met en évidence une méthode de preuve intéressante, et dont j'ai montré la pertinence dans le cadre de la vérification de la complexité de programmes. Son cadre d'application est en fait plus général, ce qui a motivé l'implémentation d'une bibliothèque réutilisable indépendante. J'en suis l'auteur et développeur principal.

La bibliothèque est documentée en détail, disponible à l'installation dans le dépot `opam` officiel pour Coq, et a reçu quelques contributions d'un utilisateur externe (Fabian Kunze, étudiant en thèse à Saarland University).

`Family=research; Audience=community; Evolution=basic; Duration=2; Contribution=leader,devel`
1 kLOC (total)

https://github.com/Armael/coq-procrastination

[16] La bibliothèque Coq `minicooper` fournit une tactique Coq pour l'élimination des quantificateurs pour l'arithmétique, grâce à une implémentation vérifiée de l'algorithme de Cooper. Comme [15], cette bibliothèque a été développée pour aider aux preuves de complexité en combinaison avec [9], mais a un cadre d'application plus général. La bibliothèque est disponible à l'installation dans le dépot `opam` pour Coq.

Je suis l'auteur principal du développement, et auteur d'environ 60% du code, une partie du code initial étant dû à François Pottier.

`Family=vehicle; Audience=community; Evolution=basic; Duration=1; Contribution=leader,devel`
4 kLOC (total)

https://github.com/Armael/minicooper

4. Impact socio-économique et transfert / *Socio-economic impact and transfer*

Intégration de l'algorithme formellement vérifié de détection de cycles dans le *build system* dune.
Le *build system* dune est l'outil actuellement le plus utilisé pour gérer la compilation des programmes et bibliothèques dans l'écosystème OCaml. Il s'agit d'un logiciel libre, activement développé et maintenu de façon communautaire (https://github.com/ocaml/dune).
En interne, dune s'appuie – entre autres – sur un algorithme de détection incrémentale de cycles dans un graphe, afin de gérer les dépendances entre les différentes actions de compilation à effectuer. J'ai intégré à dune l'implémentation de l'algorithme de détection incrémentale de cycles dont j'avais établi formellement en Coq la correction fonctionnelle et la complexité asymptotique [10]. Mon implémentation vérifiée à remplacé une implémentation non-vérifiée du même algorithme. Bien que l'implémentation précédente ait été soigneusement testée, elle contenait malgré tout plusieurs *bugs de complexité* qui dégradaient ses performances, en particulier lorsque dune était appelé sur de gros projets.
Ma contribution a été majoritairement ponctuelle : j'ai soumis une *pull request* (https://github.com/ocaml/dune/pull/1955), qui a été relue puis acceptée par Jérémie Dimino, un des auteurs de dune. Je suis également intervenu lors d'une discussion ultérieure, apportant mon expertise pour guider l'évolution du code de dune intéragissant avec mon code vérifié (https://github.com/ocaml/dune/pull/2959).
En plus du développement de la bibliothèque vérifiée elle même, mes contributions ont été d'écrire le code permettant de l'intégrer avec le reste de dune (ce qui correspond au contenu de la *pull request*), et la mesure de ses performances, permettant de constater que sur un "gros projet" réaliste, mon implémentation vérifiée était 7 fois plus rapide que celle qu'elle remplaçait.
Le résultat de ce travail est, du côté de dune, un code plus rapide et contenant moins de bugs. Du côté de ma bibliothèque vérifiée, ce travail a nécessité l'ajout de quelques fonctionnalités (qui étaient requises par le reste du code de dune), et qui sont maintenant disponibles pour l'ensemble des utilisateurs potentiels de la bibliothèque.